Today

- 2D Scan Conversion
  - Drawing Lines
  - Drawing Curves
  - Filled Polygons
  - Filling Algorithms
Drawing a Line

• Basically, it's easy... but for the details
• Lines are a basic primitive that needs to be done well...
Drawing a Line

Some things to consider

- How thick are lines?
- How should they join up?
- Which pixels are the right ones?

For example:

\[ p_1 = (x_1, y_1), \quad p_2 = (x_2, y_2) \]

\[ p_1, p_2 \text{ are inclusive endpoints} \]
Drawing a Line

\[ y = m \cdot x + b, x \in [x_1, x_2] \]

\[ m = \frac{y_2 - y_1}{x_2 - x_1} \]

\[ b = y_1 - m \cdot x_1 \]

\[ \Delta x = 1 \]

\[ \Delta y = m \cdot \Delta x \]

\[ x = x_1 \]

\[ y = y_1 \]

\[ \text{while}(x \leq x_2) \]

\[ \text{plot}(x, y) \]

\[ x++ \]

\[ y += \Delta y \]
Drawing a Line

\[ \Delta x = 1 \]
\[ \Delta y = m \cdot \Delta x \]

After rounding

Accumulation of roundoff errors

How slow is float-to-int conversion?
void drawLine(int x1, x2, int y1, y2)  
float m = (y2-y1)/(x2-x1)  
int x = x1  
float y = y1  

while (x <= x2)  
setPixel(x, round(y), PIXEL_ON)  
x += 1  
y += m
void drawLine2(int x1, x2, int y1, y2)
float m = float(y2-y1)/(x2-x1)
int x = x1
int y = y1
float e = 0.0
while (x <= x2)
    setPixel(x, y, PIXEL_ON)
    x += 1
    e += m
    if (e >= 0.5)
        y += 1
        e -= 1.0

No more rounding

void drawLine3(int x1, x2, int y1, y2)
int x = x1
int y = y1
float e = -0.5
while (x <= x2)
    set Pixel(x, y, PIXEL ON)
    x += 1
    e += float(y2-y1)/(x2-x1)
    if (e >= 0.0)
        y += 1
        e -= 1.0
### Drawing a Line

```c
void drawLine-Error4(int x1, x2, int y1, y2)
{
    int x = x1
    int y = y1
    float e = -0.5*(x2-x1)        // was -0.5
    while (x <= x2)
    {
        setPixel(x,y,PIXEL_ON)
        x += 1
        e += y2-y1                  // was /(x2-x1)
        if (e >= 0.0)               // no change
            y+=1
        e-=(x2-x1)                // was 1.0
    }
}
```

### Drawing a Line

```c
void drawLine-Error5(int x1, x2, int y1, y2)
{
    int x = x1
    int y = y1
    int e = -(x2-x1)              // removed *0.5
    while (x <= x2)
    {
        setPixel(x,y,PIXEL_ON)
        x += 1
        e += 2*(y2-y1)              // added 2*
        if (e >= 0.0)               // no change
            y+=1
        e-=2*(x2-x1)              // added 2*
    }
}
null
Drawing Curves

- Parametric curves
  - Both \( x \) and \( y \) are a function of some third parameter

\[
\begin{align*}
  x &= f(u) \\
  y &= f(u) \\
  x &= f(u) \\
  u &\in [u_0 \ldots u_1]
\end{align*}
\]
• Draw curves by drawing line segments
  • Must take care in computing end points for lines
  • How long should each line segment be?

\[ x = f(u) \quad u \in [u_0 \ldots u_1] \]
Drawing Curves

• Midpoint-test subdivision

| f(u_{mid}) - 1(0.5) |
## Drawing Curves

- **Midpoint-test subdivision**

\[ f(u_{\text{mid}}) - I(0.5) \]

- Not perfect
- We need more information for a guarantee...
Filling Triangles

- Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel(int x, int y, Color rgba)
```

- Example: Filling the inside of a triangle
• Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Antialiased edges
  - No cracks between adjacent primitives
  - MUST BE FAST!
Simple Algorithm

- Color all pixels inside triangle

```c
void ScanTriangle(Triangle T, Color rgba) {
    for each pixel P at (x, y) {
        if (Inside(T, P)) {
            SetPixel(x, y, rgba);
        }
    }
}
```

Line Defines Two Halfspaces

- Implicit equation for a line
  - On line: \( ax + by + c = 0 \)
  - On right: \( ax + by + c < 0 \)
  - On left: \( ax + by + c > 0 \)
Inside Triangle Test

- Point is inside triangle if it is in positive halfspace of all three boundary lines
  - Triangle vertices are ordered counter-clockwise
  - Point must be on the left side of every boundary line

Boolean Inside(Triangle T, Point P)
{
    for each boundary line L of T {
        Scalar d = L.a*x + L.b*y + L.c;
        if (d < 0.0) return FALSE;
    }
    return TRUE;
}
Simple Algorithm

- What is bad about this algorithm?

```c
void ScanTriangle(Triangle T, Color rgba)
{
    for each pixel P at (x,y)
    {
        if (Inside(T, P))
            SetPixel(x, y, rgb);  
    }
}
```

Optimize for Triangles

- Split triangle into two parts
  - Two edges per part
  - Y-span is monotonic
- For each row
  - Interpolate span
  - Interpolate barycentric coordinates
Triangle Sweep-Line Algorithm

• Take advantage of spatial coherence
  - Compute which pixels are inside using horizontal spans
  - Process horizontal spans in scan-line order
• Take advantage of edge linearity
  - Use edge slopes to update coordinates incrementally

```c
void ScanTriangle(Triangle T, Color rgba){
  for each edge pair {
    initialize x_L, x_R;
    compute dx_L/dy_L and dx_R/dy_R;
    for each scanline at y
      for (int x = ceil(x_L); x <= x_R; x++)
        SetPixel(x, y, rgba);
    x_L += dx_L/dy_L;
    x_R += dx_R/dy_R;
  }
}
```

Bresenham’s algorithm works the same way, but uses only integer operations!
Antialiasing

Desired solution of an integral over pixel

Hardware Antialiasing

Supersample pixels
- Multiple samples per pixel
- Average subpixel intensities (box filter)
- Trades intensity resolution for spatial resolution
Hardware Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles

Polygon Scan Conversion

- Fill pixels inside a polygon
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting
  - Holes

What problems do we encounter with arbitrary polygons?
Polygon Scan Conversion

- Need better test for points inside polygon
  - Triangle method works only for convex polygons

**Convex Polygon**

**Concave Polygon**

Inside Polygon Rule

- What is a good rule for which pixels are inside?

- Concave
- Self-Intersecting
- With Holes
Inside Polygon Rule

- Odd-parity rule
  - Any ray from P to infinity crosses odd number of edges

Inside/Outside Testing

- The Polygon
- Non-exterior
- Non-zero winding
- Parity
Filled Polygons

Filled Polygons
Filled Polygons

What happens at these locations?

Filled Polygons

If we count ONCE...
If we count TWICE...
Filled Polygons

- "Equality Removal" applies to all vertices
- Both x and y coordinates

Final result:
Filled Polygons

• Who does this pixel belong to?

Drawing a Line

• How thick?

• Ends:

  Butt
  Round
  Square
Flood Fill

Span-Based Algorithm

Definition: a **run** is a horizontal span of identically colored pixels

1. Start at pixel “s”, the seed.
2. Find the run containing “s” (“b” to “a”).
3. Fill that run with the new color.
4. Search every pixel above run, looking for pixels of interior color
5. For each one found,
   6. Find left side of that run (“c”), and push that on a stack.
7. Repeat lines 4-7 for the pixels below (“d”).
8. Pop stack and repeat procedure with the new seed.

The algorithm finds runs ending at “e”, “f”, “g”, “h”, and “i”